Attorney Docket No. 07844/28001
Client No. P254

APPENDIX

APPENDIX

This appendix contains exemplary implementations of a system and a method for user interface substitution using Java. First, one exemplary Java code for a map base class is shown below:

```
/**
 * Interface implemented by each Map class.
 * <p>
 * This interface will grow as new interactions between Maps and Fashions
 * are defined.
 *
 * @version    %I%, %G%
 * @author     Chansler
 */
interface Map {
  /**
   * set the UI for this Map
   *
   * @param   ui  UI that created this Map
   */
  void setUI(UI ui);
  /**
   * get the UI for this Map
   *
   * @return  UI  Ui that created this Map
   */
  UI getUI();
  /**
   * Notify Map that the parent UI is quitting.
   */
  void dispose();
  /**
   * Notify Map that an action occurred in a choice tree.
   * This is a direct call from the Fashion.
   * (may have been more stylish to have used event registration.)
   * Clues are analyzed, and a dispatch is performed. If the action
   * cannot be completed locally, Preferences are set, and the UI is
   * notified. The UI is expected to generate a UIEvent and the Main
   * will conditionally restart the UI.
   *
   * @param group    id of choice group
   * @param item     id of chosen item from group
   * @param clue     clue that identifies kind of choice group
```

```
     * @param selected  indicates state of bi-stable choice item
     */
    void treeAction(String group, String item,
                    String clue, Boolean selected);
    /**
     * Navigate to the ultimate beginning
     */
    void begin();
    /**
     * Navigate to the ultimate end
     */
    void end();
    /**
     * Navigate to the local beginning
     */
    void first();
    /**
     * Navigate to the local end
     */
    void last();
    /**
     * Navigate to the local previous
     */
    void previous();
    /**
     * Navigate to the local next
     */
    void next();
    /**
     * Navigate by name
     *
     * @param where name of UI to go to
     * @param first do a "first" when you get there
     * @param last  do a "last" when you get there
     */
    void navigate(String where, Boolean first, Boolean last);
}
```

Exemplary Java code for a fashion base class is shown below:

```
/**
 * Interface implemented by each Fashion class.
 * <p>
 * This interface will grow as new interactions between Maps and Fashions
 * are defined.
```

```
 *
 * @version    %I%, %G%
 * @author     Chansler
 */
interface Fashion {
  /**
   * set the UI for this Fashion
   *
   * @param   ui  UI that created this Fashion
   */
  void setUI(UI ui);
  /**
   * get the UI for this Fashion
   *
   * @return  UI  Ui that created this Fashion
   */
  UI getUI();
  /**
   * Notify Fashion that the parent UI is quitting.
   */
  void dispose();
  /**
   * Get subordinate container, if any.
   */
  java.awt.Container nextContainer();
  /**
   * Get subordinate container, if any.
   *
   * @param key   id to select in Fashion's layout manager
   */
  java.awt.Container nextContainer(String key);
  /**
   * Get subordinate container, if any, from the i'th parallel display.
   *
   * @param i     selects the i'th parallel display
   * @param key   id to select in Fashion's layout manager
   */
  java.awt.Container nextContainer(int i, String key);
  /**
   * present a choice tree.
   *
   * @param   id      key of root description
   *
   * @return  Object  handle for choice tree
   */
  Object tree(String id);
```

```
/**
 * present a message to the user
 *
 * @param   group      choice group
 * @param   item       item from group identifying message
 */
void message(String group, String item);
/**
 * present a message to the user
 *
 * @param   group      choice group
 * @param   item       item from group identifying message
 * @param   tail       extra run-time information
 */
void message(String group, String item, String tail);
/**
 * request confirmation of choice from the user
 *
 * @param   group      choice group
 * @param   item       item from group identifying message
 *
 * @return  Boolean    whether the user confirmed choice
 */
Boolean confirm(String group, String item);
/**
 * request a file name from the user
 *
 * @param   group      choice group
 * @param   item       item from group identifying message
 *
 * @return  String     name of file chosen, or else null if canceled
 */
String file(String group, String item);
/**
 * push a parameter list to the UI
 *
 * @param   item    name of parameter list
 * @param   value   parameter list
 *
 */
public void setParams(String item, Hashtable value);
/**
 * get a parameter list from the UI
 *
 * @param   item    name of parameter list
 *
```

```
     * @return  Hashtable   parameter list
     */
    Hashtable getParams(String item);
    /**
     * make component visible (or not)
     *
     * @param   key     name of component
     * @param   on      visible or not
     */
    void setVisible(String key, boolean on);
    /**
     * make component visible (or not)
     *
     * @param   i       selects the i'th parallel display
     * @param   key     name of component
     * @param   on      visible or not
     */
    void setVisible(int i, String key, boolean on);
    /**
     * request parallel displays from Fashion
     */
    void split(int i);
}
```

In general, the methods getUI, setUI and dispose connect the Fashion object to its UI object. The methods nextContainer and split organize the allocation of resources among Fashion object. The methods file, confirm, and message communicate with the user. The method tree is used to present interface items/widgets. The method setVisible request the presentation of the user interface components controlled by this Fashion object.

Code for UI is shown next. The UI is the glue that binds together a Map with logic and a Fashion with knowledge of presentation. In addition, a SafeBundle supplies descriptive information interpreted by the Map and Fashion.

```
     * A UI has some obvious members:
       /**
        * current Map object
```

```
 */
Map map;
/**
 * current Fashion object
 */
Fashion fashion;
/**
 * current ResourceBundle with Map/Fashion data
 */
ResourceBundle description;
/**
 * parent UI
 */
 UI mother;
```

```
* The construction parameters for a new UI include:
   * @param      mother      the parent of this
   * @param      mapName     name of Map class
   * @param      fashionName name of Fashion class
   * @param      box         Container for presenting UI
```

Pseudo-code for the constructor code is shown below:

1. Use the mapName and fashionName to lookup the ResourceBundle description.

2. Create new map and fashion objects by name (Java:class.forName(mapName).newInstance())

3. Invoke the setUI method of both the map and fashion objects with this UI as a parameter. That is, each Map and Fashion belong to a UI that provides some communication logic. This method of Map classes can construct subordinate UI objects, for which this UI will be the parent. Other methods allow Maps to register with the parent UI to receive notification of events associated with user interactions. Other methods provide for restarting and termination of the user interface.

Another exemplary Fashion class is used for the typical interface display. The class can instantiate the conventional interface widgets such as buttons, labels, text areas,

and others. When the Map asks that a display be presented (the method tree), this Fashion gets a description of the interface from the ResouceBundle of its UI parent. This description specifies what choices and information are to be displayed for the user.

Each piece of the display is composed of "parts" and each part may itself be an additional collection of parts. The main logic of this class (buildTrees and buildSubTrees) recursively analyze the description of parts.

For each primitive part, its description tells what kind of interface primitive to use (a button, for instance) and specifies parameters for the primitive (the button's label, and what icon should decorate the button when it is pushed, for instance).

For parts with parts, the description tells how to layout the subparts (whether a list of buttons is to be stacked horizontally or vertically, for instance). For each interface primitive that has an action associated with it (pressing a button, for instance) the action event is set to call the treeAction method of the Map of the parent UI.

A class for UI objects with map and fashion members is shown next:

```
/**
 * A UI is the glue that binds together a Map with logic and a Fashion
 * with knowledge of presentation. In addition, a SafeBundle supplies
 * descriptive information interpreted by the Map and Fashion. The Map and
 * the Fashion share the Preferences and MessageLog held by the UI object.
 * <p>
 * This UI class will query the Map for instructions on creating a
 * subordinate UI object, if required.  The UI's are linked as
 * <code>parent</code> and <code>daughter</code>.
 * <p>
 * Map and Fashion objects are constructed by the UI by loading and
 * instantiating the proper classes by name via the intelligent cache.
 * The SafeBundle is also supplied by the cache.
 *
 * @version    %I%, %G%
 * @author     Chansler
 */
class UI {
```

```
/**
 * registry for all 'action' buttons created by fashions.
 * entries should be JButtons.
 */
private static final Hashtable actionRegistry = new Hashtable(47);
/**
 * label position in parsed description
 */
protected static final int LABEL = 0;
/**
 * clue position in parsed description
 */
protected static final int CLUE = 1;
/**
 * accessibility position in parsed description
 */
protected static final int ACCESS = 2;
/**
 * parts position in parsed description
 */
protected static final int PARTS = 3;
/**
 * key position in a label
 */
protected static final int KEYPOS = 0;
/**
 * position of remainder of label
 */
protected static final int LABELPOS = 2;
/**
 * syntax terminal for parsing descriptions
 */
protected static final char PLANSEP = ';';
/**
 * key in resource bundle for the root of choice tree
 */
protected static final String ROOT = "ROOT";
/**
 * key to look up window title
 */
protected static final String DESC = "DESC";
/**
 * key to look up application title
 */
protected static final String TITLE = "TITLE";
/**
```

```
 * text of clue in description resource that identifies a on/off item
 */
protected static final String OPTION_CLUE = "option";
/**
 * key suffix to look up dialog text
 */
protected static final String TEXT_PART = "-text";
/**
 * key to look up query window title
 */
protected static final String QUERY = "QUERY";
/**
 * key to look up message window title
 */
protected static final String MESG = "MESG";

/*
 * These instance variables have package scope so that they can be
 * conveniently shared by the Map and the Fashion. If protected,
    • <code>get</code>, but not <code>set</code> methods would be
    • required.
 */
/**
 * name of Map
 */
String mapName;
/**
 * name of Fashion
 */
String fashionName;
/**
 * current Preferences
 */
Preferences preferences;
/**
 * current diagnostic message facility
 */
MessageLog messageLog;
/**
 * current connection to server
 */
IntelligentCache intelligentCache;
/**
 * current Container for UI presentation
 */
Container container;
```

```
/**
 * current Map object
 */
Map map;
/**
 * current Fashion object
 */
Fashion fashion;
/**
 * current ResourceBundle with Map/Fashion data
 */
SafeBundle description;
/**
 * parent UI
 */
 UI mother;
/** global resources */
static Globals globals;
static Geoff geoff;

/**
 * queue of listener for UIEvents from this object
 */
protected EventListenerList listenerList = new EventListenerList();
/**
 * general constructor parameters for all imported objects
 *
 * @param      mother      the parent of this
 * @param      mapName     name of Map class
 * @param      fashionName name of Fashion class
 * @param      pref        current Preferences
 * @param      log         diagnostic message facility
 * @param      cache       current intelligent cache
 * @param      box         Container for presenting UI
 *
 * @return     UI
 */
UI(UI mother,
      String mapName,
      String fashionName,
      Preferences pref,
      MessageLog log,
      IntelligentCache cache,
      Container box){
   this.mother = mother;
   if (mother == null) {
```

```
        globals = new Globals();
        globals.setPromiscuous("UI.globals");
        geoff = new Geoff(log, pref, cache);
    }
    this.fashionName = (fashionName == null) ? "FashionNull":fashionName;
    this.mapName = (mapName == null) ? "MapNull":mapName;
    preferences = pref;
    messageLog = log;
    intelligentCache = cache;
    container = box;
    Trace.traceTemp("new UI(" + mapName + ", " + fashionName + ")");

    description = cache.newBundle(this.mapName, this.fashionName);
    listenToWindow();

    map = (Map)cache.newObject(Map.class, this.mapName);
    fashion = (Fashion)cache.newObject(Fashion.class, this.fashionName);

    if (fashion != null){
        fashion.setUI(this);
    } else {
        messageLog.panic(this, "failed to create fashion " + this.fashionName);
        return;
    }

    if (map != null){
        map.setUI(this);
    } else {
        messageLog.panic(this, "failed to create map " + this.mapName);
        return;
    }
}
/**
 * request to quit (from Map); fires Quit UIEvent
 *
 * @param   why     identification of request source
 */
protected void quit(Object why){
    messageLog.log(this, "Good bye! Thanks for all the fish! " + why);
    fireUIEvent(new UIEvent(this, UIEvent.UIQuit));
}
/**
 * Request to restart (from Map); fires Restart UIEvent.
 * Resart is conditioned upon there being a real need. Generally
 * requested when personality parameters are suspected of having been
 * changed.
```

```
 *
 * @param   why    identification of request source
 */
protected void restart(Object why){
   messageLog.log(this, "Restarting UI because of " + why);
   fireUIEvent(new UIEvent(this, UIEvent.UIRestart));
}
/**
 * listen to window event; called if container is a JFrame
 */
protected void listenToWindow(){
   if ( ! (container instanceof JFrame))
      return;
   JFrame jframe = (JFrame)container;

         WindowListener listen = new WindowAdapter() {
            public void windowClosing(WindowEvent e) {quit(this);}
         };
         jframe.addWindowListener(listen);
      }
      /**
       * make UI go away
       */
      public void dispose(){
         if (map != null) map.dispose();
         if (fashion != null) fashion.dispose();
      }

/**
 * Adds a UIListener to the UI.
 */
public void addUIListener(UIListener listener){
   listenerList.add(UIListener.class, listener);
}

/**
 * Removes a UIListener from the UI.
 */
public void removeUIListener(UIListener listener) {
         listenerList.remove(UIListener.class, listener);
}

/*
 * Notify all listeners that have registered interest for
 * notification on this event type.
 * (Borrowed from swing implementation, without lazy events. -RC)
```

```
 * @see EventListenerList
 */
protected void fireUIEvent(UIEvent event) {
      // Guaranteed to return a non-null array
      Object[] listeners = listenerList.getListenerList();
      // Process the listeners last to first, notifying
      // those that are interested in this event
      for (int i = listeners.length-2; i>=0; i-=2) {
        if (listeners[i]==UIListener.class) {
              ((UIListener)listeners[i+1]).UIAction(event);
        }
      }
}
/**
 * Construct a combination record for use as an actionCommand.
 * (must necessarily be a String and not a simple record class)
 */
String actionCommand(String parent, String part, String clue){
   return parent + UI.PLANSEP + part + UI.PLANSEP + clue;
}

/**
 * Define a shared notion of "selected" for different preference items.
 *
 * @param id    typically the actionCommand for node
 *
 * @return boolean
 */
protected boolean selected(String id){
   String[] details = StringUtils.string2Array(id, UI.PLANSEP);
   return (OPTION_CLUE.equals(details[2]))
      ? preferences.getBooleanValue(details[1], false)
      : preferences.sameas(details[0], details[1]);
}

void registerAction(String actionID, JButton action) {
   if (actionRegistry.put(actionID, action) != null)
      messageLog.panic(this, "Action replaced in registry!");
}

void setActionEnabled(String actionID, boolean enabled) {
   JButton jb = (JButton) actionRegistry.get(actionID);
   if (jb == null) {
      messageLog.panic(this, "(dis/en)abling non-existent action!");
   } else {
      jb.setEnabled(enabled);
```

```
    }
  }
}
```

An exemplary top-level map called "MapZero" that implements the top level menus is discussed next. This implementation of the Map interface is the Map at the root of the UI tree.

```
MapZero functions:
* <ul>
* <li>select Locale from list</li>
* <li>select Fashion from list</li>
* <li>select Map from list</li>
* <li>select Look'n'Feel from list</li>
* <li>select whether to use ToolTips</li>
* <li>present Help and About message dialogs</li>
* <li>load/save Preferences</li>
* <li>exit UI</li>
* </ul>
* To change personality parameters, the Map saves the new selection in
* the Preferences, and requests that the UI restart. If the Help or About
* dialogs are to be presented, the Map makes a direct request of the
* Fasion. The Map makes direct requests to turn ToolTips on and off.
* If application exit is requested, the Map notifies the UI.
* <p>
* This Map relys on an appropriate resourceBundle.
```

When the parent UI constructs the MapZero object and calls the setUI method of this class. That method constructs the one subordinate UI: (A more complex Map would do this for each subordinate UI.)

1. Get the name of the next Map from the ResourceBundle of the parent UI.
2. Get the name of the next Fashion from the ResourceBundle of the parent UI.
3. Ask the Fashion of this UI for the display resouces (nextContainer) to be used for the new UI.
4. Construct the new UI.
5. Register for event notifications from the new UI.

The business logic of this map manages the application preferences. The treeAction method of this class just sets the appropriate parameters.